VMM Verification Methodology Manual 活用テクニック

赤星博輝

第3回　ランダム生成の機能を使いこなそう

検証ライブラリとその利用ガイドラインである“Verification Methodology Manual for SystemVerilog(VMM)”の活用法を解説する連載の第3回である．今回は，vmm_atomic_genを中心としたランダム生成について解説する．VMMのランダム生成とSystemVerilogの機能を使い，さまざまなテスト・パターンを効率的に作成する．（編集部）

新しい技術を導入しようとすると，いろいろな問題(壁)が発生し，その壁を乗り越える必要があります．現在の状態と理想とする状態の差が大きければ大きいほど，その壁は高くなります．

● VMM導入の三つの壁

これまでの経験から，VMMの壁は**図1**に示すように三つの要因が組み合わさってできていると感じています．

1）新しい言語であるSystemVerilogに対するギャップ
2）新しい検証ライブラリであるVMM標準ライブラリに対するギャップ
3）新しい考え方であるVerification Methodologyに対するギャップ

SystemVerilogでは，検証に関する機能が大幅に向上しています．また，オブジェクト指向プログラミングが可能になりました．これまでのVerilog HDLやVHDLといったハードウェア記述言語は，オブジェクト指向言語ではありません．このため，ハードウェア設計者はこれまで使ったことがない継承やオーバロード，オーバライドといった機能を使う必要が出てきます．C++やJavaなどに慣れた技術者であれば既に使い慣れた機能です．しかし，ハードウェア系の技術者の場合にはこれまで，C++やJavaを用いてオブジェクト指向でプログラミングする機会が少なく，導入に対する大きなギャップになっています．

次に越えるべきギャップは新しいライブラリです．初めてのものは誰でも戸惑うものです．この点については場数を踏むことで慣れる必要があります．

そして，一番問題になるものは，新しい考え方です．VMMはオブジェクト指向の技術を使い，検証に必要なライブラリを構築し，最小のコーディング量で最大の検証パターンを生成することを目標にしています．この目標を満たすため，VMMにはいろいろなルールが記述されています．ただし，ルールの数が多いため，すべて覚えておくことは難しいと思います．ルールを覚えるよりは，その背景を理解することが重要になります．

この三つの壁はそれなりに高いものだと思いますが，逆に乗り越えてしまえば，検証効率を大幅に改善することが
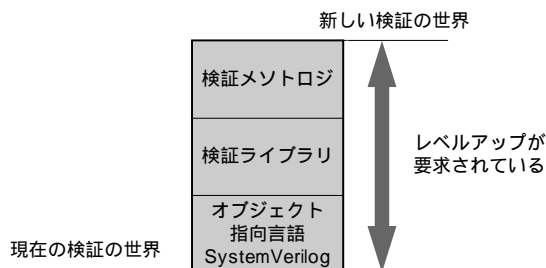
図1　VMMを導入する際の三つの壁
既存のHDLベースのテストベンチ環境からVMMに移行するには三つの壁があるので，一つずつ壁を乗り越えていく必要がある．

KeyWord　SystemVerirog，VMM，ランダム生成，vmm_atomic_gen，vmm_channnel，vmm_xactor，vmm_env，vmm_data，rand，rand_mode，テスト・パターン，constraint_mode

できます．ぜひ，この機会にVMMをマスタしてしまいましょう．

● 少ない記述量でランダム値を生成できる

前回（本誌2006年10月号，pp.139-147を参照）は，**図2**のような2次元データの領域判定を行う回路の検証を行いました．VMMのライブラリを使用し，**図3**のようにランダム生成（vmm_atomic_gen），チャネル（vmm_channel），トランザクタ（vmm_xactor），制御（vmm_env）を行うテストベンチ環境を作成しました．今回は，vmm_atomic_genを中心としたランダム生成について解説します．

ここで前回の復習をしましょう．まず，データ用のクラスを定義するときには，vmm_dataを継承して作成します．あらかじめマクロが用意されており，このデータ型を用いてマクロを呼び出すことでランダム生成を行うクラスを作成できます．少ない記述量でランダム値を生成できることがVMMのランダム生成の特徴といえます．
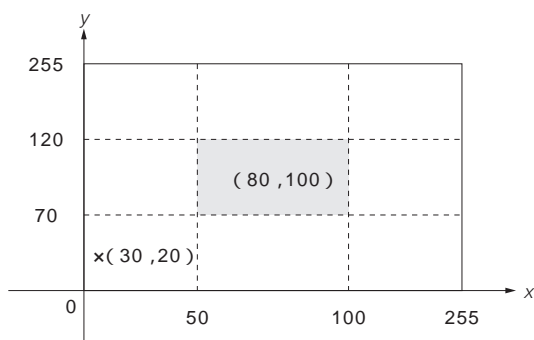
図2　ターゲットの判定回路の動き

256×256の中に（50，70）～（100，120）の長方形があり，その領域内であれば'1'，領域外であれば'0'と判定する．
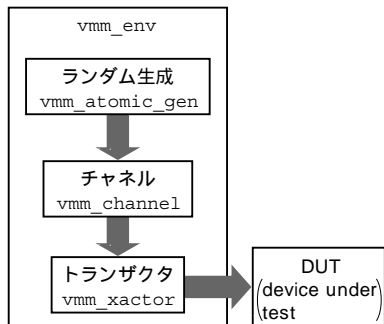
図3　前回の構成と今回の注目点

前回はランダム生成，チャネル，トランザクタ，制御について概要を説明したが，今回はランダム生成について詳細を説明する．

```
class xy_dat extends vmm_data;
//省略
endclass
// データ型定義
'vmm_atomic_gen(xy_data)
// xy_dataのランダム生成クラス定義
```

今回は，VMMのランダム生成とSystemVerilogの機能を使って，さまざまなテスト・パターンを効率的に作成する方法について紹介します．

一部の変数のランダム生成を停止する

図2のような2次元データ（mX，mY）で考えてみます．二つの変数を同時にランダム生成せずに，mXの値は固定にして，mYのみをランダム生成することで，特定の状態についてチェックしたい場合があります（ある特定の縦方向のチェックを行うケース）．このとき，新しいクラスを作って，mYのみランダム変数とすることができます．しかし，個別にクラスを作っていくと，**表1**に示すように組み合わせの数だけクラスができることになります．今回は変数が2個なので四つの組み合わせで済みますが，8変数なら256，16変数なら65536もの組み合わせを使う可能性があります．これでは，最小の記述量で最大のテスト・パターンを発生させるというVMMの目標からは，遠ざかっているといえます．

● randによるランダム生成のON/OFF

SystemVerilogでは，randというアトリビュートを付けた変数に対して，rand_modeというメソッドを使用して，ランダム生成をONしたり，OFFしたりできます．この機能を利用し，状況に応じてランダム生成を制御できます．

2次元のデータ（mX，mY）のmXに対してランダム生成を

表1　アトリビュートの組み合わせ

変数が増加すると，randあり・なしの組み合わせは，指数的に増加してしまう．そのため，VMMではrandを付けることを推奨している．

	mX	mY
Case 1	randなし	randなし
Case 2	randなし	randあり
Case 3	randあり	randなし
Case 4	randあり	randあり

リスト1　ランダム生成をON/OFFする
SystemVerilogで特定の変数のランダム生成をOFFするには，変数に対して，rand_mode(0)とする．ランダム生成をONにするには，変数に対して，rand_mode(1)とする．

```
class xy_dat  extends  vmm_data;
    rand logic[7:0] mX,mY;
   ・・・
endclass

xy_dat p=new;

Initial begin
 ・・・
p.mX.rand_mode(0); // mXのランダム生成OFF
p.randomize( );
 ・・・
p.mX.rand_mode(1) ; // mXのランダム生成ON
p.randomize( );
 ・・・
end
```

リスト2　vmm_atomic_genでのランダム生成のON/OFF
特定の変数のランダム生成をOFFするには，randomized_objの変数に対してrand_mode(0)を実行する．

```
class xy_dat_atomic_gen extends...;

     xy_dat     randomized_obj;

endclass
 ・・・・
 xy_dat_atomic_gen xy_gen;

 virtual task reset_dut();
    super.reset_dut();
    // mXの設定
    xy_gen.randomized_obj.mX =10;
    xy_gen.randomized_obj.mX.rand_mode(0);
 endtask
```


ON/OFFする記述を**リスト1**に示します．mXに対してrand_mode(0)とすることでランダム生成をOFFに，rand_mode(1)とすることでランダム生成をONにできます．この場合はmXの変数に対してのみON/OFFを行いましたが，mYに対しても個別にランダム生成のON/OFFを行えます．

● vmm_atomic_genによるランダム生成のON/OFF

SystemVerilogだけの世界からVMMを使っている場合に話を進めていきます．VMMはSystemVerilogで実装されているので，VMMのランダム生成を行う場合でもSystemVerilogの機能を使って，ランダム生成を変数ごとにOFFしたり，またOFFしたものをONすることができます．

vmm_atomic_genを使ってランダム生成している場合，ランダム生成のON/OFFはどうしたらよいのでしょうか．マクロで定義されるため，どのようにしてランダム生成をON/OFFしてよいか分からないと思われるかもしれません．

こうした場合にはVMMのテキスト(1)を参照する必要があります．VMMのAppendix A-10にvmm_atomic_genに関する説明があります．データ・クラスのための変数としてrandomized_objが使用されていることが分かります．'vmm_atomic_gen(xy_dat)によって作成されたxy_dat_atomic_genのクラスに対して，xy_datの変数mXの値を10に設定し，そのmXのランダム生成をOFFするための記述は，**リスト2**のようになります．

また，この記述は，vmm_envの実行シーケンスの中でreset_dut( )に記述しているところがポイントです．reset_dutはvmm_envで規定されている実行シーケンスの一つで，リセット時に行う処理を記述するフェーズになります．前回のテストベンチに対して，reset_dutに今回のランダム生成をOFFする記述を追加するだけで，これまで2次元に散らばっていたテスト・パターンを，(X座標が固定の)1次元に散らばるテスト・パターンに変更できます．

● ランダム生成の設定の変更は実行シーケンスでできる

VMMのポイントの一つは，vmm_envの実行シーケンスで設定を変更することで，これまでとは違ったテスト・パターンを生成できることにあります．これを実際の検証現場で考えてみましょう．例えば，最初に広範囲のランダム生成を用いたテストベンチを作成したとします．その後，コーナ・ケースに注目したテストを行うため，ランダム生成をOFFにしてより狭い範囲のパターンを生成する場合など，その変更はvmm_envが用意したreset_dutというタスクに記述を追加するだけで可能になります．

ここでは，テスト・パターンを変更するときに，いちいち最下層のクラスxy_datを変更するのではなく，検証レイヤの最上位であるvmm_envのクラスだけを変更すればよいことがポイントです．変更個所が大変少なく，多くのテスト・パターンを流すときには有利になります．

VMMのルール4-59に以下のような記述があります．

ルール4-59：プロトコル・プロパティやフィールドに対応するクラス・プロパティは，すべてrandアトリビュートを持つものとする．

randアトリビュートを付けておけば，rand_modeを用いていつでもランダム生成をOFFできるため，ON/OFFどちらにも対応できるようになります．これにより，テストベンチの再利用が容易になるためです．

例えば，最初はmX＝100でmYを網羅的にテストするテスト・パターンを生成するだけでよい場合でも，その後，mYが固定値でmXの値もランダム生成を行ったりする状況が発生したり，mX，mYを同時ランダム生成したい状況が発生する場合が考えられます．

ルール4-59を守っておけば，すべてのケースに容易に対応できます．VMMにはこのような知識がちりばめられており，検証エンジニアには非常に有用な書籍といえます．

制約を使ったランダム生成

SystemVerilogでは，ランダム生成をOFFするだけでなく，ランダム変数のとる値に制約を与えることができます．

一般に入力として与えられるパターンは，偏りがあったり，入力として発生することがないパターンがあります．そのため，シミュレーション時になんらかの制約を与えることができないと，本来発生しないパターンをシミュレーションすることになります．意味のないシミュレーションを行えば，無駄に計算機パワーを使うことになります．

● constraintで制約を与える

制約を与えるには，データを扱うためにvmm_dataを継承して作成したクラスで，constraintを用いてランダム生成時に守るべきルールを記述します．**リスト3**にmXとmYの値が同じという制約を記述した例を示します．

こうした制約を与えることで，単純なランダム生成ではあまり発生しない状況を重点的に与えることができるようになります．

● 複数の制約を使ったランダム生成

さらに，SystemVerilogでは複数の制約を定義することができます．

複雑な制約を一つの制約で記述することは難しいケースが少なくありません．そのような場合でも分割して複数の制約にすることで簡単に制約を記述することがあります．**リスト4**に二つの制約を与えた例を示します．

● 制約をON/OFFしてランダム生成

複数の制約を記述できるということは，**リスト4**のように二つの制約に矛盾がないものだけでなく，**リスト5**のxy_datにあるような相反する制約を与えることもできることになります．ランダム生成を実行するときに相反する制約があると，ランダム生成に失敗して，シミュレーションを進めることができなくなります．

それでは，**リスト5**のように相反する制約を書かないのかというと，決してそういうことではありません．SystemVerilogでは，制約をONしたりOFFしたりする機能があるので，この機能を使うことで相反する制約を有効に使うことができます．そのときに使用するのが，constraint_mode( )というメソッドになります．

リスト5の例では，VMMの実行シーケンスのreset_dut時に，制約CAをconstraint_mode(0)を使うことでOFFしています．これを実行すると，制約CBだけを利用してランダム生成をすることができます．

また，このconstraint_modeはシミュレーションの

リスト3　制約を与えた例

mXとmYの値は同じという制約(test_constraintA)を記述した．この制約はランダム生成時に使用される．

```
class xy_dat extends vmm_data;
 rand logic[7:0] mX,mY;
 static vmm_log log=new("XY_dat", "class");
 function new( );
   super.new(log);
 endfunction

 constraint test_constraintA { mX == mY;}

  . . .
endclass
```

リスト4　複数の制約を与えた例

test_constraintAとtest_constraintBのように複数の制約を与えることができる．複雑な制約は，いくつかの制約を分けて記述する方が楽に書ける．

```
class xy_dat extends vmm_data;
 rand logic[7:0] mX,mY;
 static vmm_log log=new("XY_dat", "class");
 function new( );
   super.new(log);
 endfunction

 constraint test_constraintA { mX == mY;}
 constraint test_constraintB { mX  <  20;}
  . . .
endclass
```

途中でも実行できるので，異なった制約を切り替えながらシミュレーションを実行できます．

制約により，さまざまな状況のテスト・パターンを発生できるため，いろいろな状況を発生させる制約をあらかじめ全部，リストアップできれば，ランダム生成を使って効果的に検証を進めることができます．

もちろん，これは理想的な状況の話で，実際には後からいろいろな要求が出てくることは日ごろの仕事でも皆さん経験されていることでしょう．そうした場合でも，新しい制約を簡単に追加できることがVMM（というかSystem Verilog）の特徴になります．

ここでは，継承を使って新しい制約を記述する方法と制約だけを別に記述する方法を紹介します．

● データ・クラスの継承を利用して制約を追加

SystemVerilogでは，継承をすることで，既存のクラスに対して機能の追加や変更を行えますが，今回は継承したクラスで制約を追加してみます．

リスト1のクラスxy_datは制約がないものでしたが，そのクラスを継承し制約を持つクラスxy_datCAを作成したのが**リスト6**です．一見すると制約しか記述していないように見えますが，extends xy_datということで，クラスxy_datの要素を継承して，さらにその制約を追加しています．

そのxy_datCAという新しいクラスでvmm_atomic_genを使う場合には，どうしたらよいでしょうか．一つのやり方として，マクロにより'vmm_atomic_gen(xy_datCA)を定義し使用する方法がありますが，xy_datCAはxy_datから継承して作成したクラスなので'vmm_atomic_gen(xy_dat)によって作成されたxy_dat_atomic_genを活用します．

xy_dat_atomic_genのクラスには，先ほども登場したxy_datを指している変数randomized_objがあります．このrandomized_objの指している先を新しいクラスに変更すると，新しいクラスを用いてランダム生成をすることができます．**リスト6**に継承したクラスをランダム生成に使用するための記述を示します．変更する際にはvmm_envのbuildの実行で，ランダム生成xy_dat_atomic_genの変数xy_genのインスタンスを作成し，そのxy_gen.randomized_objに新しいxy_datCAのインスタンスを代入しています．これにより，このxy_genはxy_datCAのクラスとしてランダム生成を行います．

このポイントは，randomized_objに代入できるのはxy_datもしくは継承したクラスだけということです．クラスの継承を使うことで変更や追加する差分だけ実装するだけよく，テストベンチの構成もほとんど再利用することができます．

● 外部の制約を利用する

リスト7に示すように，SystemVerilogでは制約の定義をクラス定義の外で行えます．クラスの内部では，中身のない制約（未定義の制約）CAとCBを定義して，クラス定義の外にCAの制約を定義することができます．ここで重要なポイントとしては，クラスの中では制約はCAとCBの二つあると定義したにもかかわらず，実際のクラス定義の外

リスト5　矛盾する複数の制約を与えた例

CAとCBの制約を同時に成立させることはできない．しかし，制約をOFFすることが可能なので，うまく活用すると効率良くシミュレーションを流すことができる．

```
class xy_dat extends vmm_data;
 rand logic[7:0] mX,mY;
 static vmm_log log=new("XY_dat", "class");
 function new( );
   super.new(log);
 endfunction

 constraint CA { mX == mY;}
 constraint CB { mY > mX;}
 . . .
endclass

 virtual task reset_dut();
    super.reset_dut();
    xy_gen.randomized_obj.CA.constraint_mode(0);
 endtask
```

リスト6　継承を用いた制約の追加

クラスの継承を用いることで，最小限の変更で制約を変更できる．VMMのランダム生成を使う場合には，randomized_objを新しいクラスのインスタンスに置き換えるだけでその変更を利用できる．

```
class xy_datCA extends xy_dat;
 constraint CA { mX == mY;}
endclass


virtual function void build();
    xy_datCA robj=new;
    super.build();
    g_d_chan=new("Gen_drv_channel","u0");
    xy_gen  =new("u1",,g_d_chan);
    xy_gen.randomized_obj= robj;
    xy_drv = new("u2",,g_d_chan);
 endfunction
```

ではCAの制約のみを定義していることです．この場合は，CBの制約は実際にはないことと同じになります．

これにより，クラスを作成するときには，常に未定義の制約を数個定義しておけば，後で制約を追加することが可能になります．

VMMのルールにも制約に関するものがあります．

推奨4-86 “test_constraintsX”という名まえの未定義のconstraintブロックを宣言する．

これは，test_constraints1，test_constraints2，…と複数個の未定義の制約を準備しておけば，クラスを作成する時点で制約があるなしにかかわらず，後で制約を追加できるために推奨されています．そのため，この外部に制約を記述する方法を使うと，制約だけを集めたファイルをクラスの定義とは別に作成できます．

この外部制約を別のファイルに記述するというやり方は，制約変更の手間が少なく，管理もしやすい方法だと思います．**図4**のように，制約ごとに異なる制約のファイルを用意しておけば，コンパイル時に切り替えることで，そのほかの部分は変更なしに異なった制約のテスト・パターンを生成できます．

リスト7　外部に制約を追加

SystemVerilogではクラスの中で制約CA，CBがあることだけを宣言し，外部に制約を記述することができる．

```
class xy_dat extends vmm_data;
 rand logic[7:0] mX,mY;
 static vmm_log log=new("XY_dat", "class");

 constraint CA;
 constraint CB;
 ・・・
endclass

constraint xy_dat::CA {mX==mY;}
```

ランダム生成の活用術

ランダム生成だけで検証がすべて終わるなら，大変うれしいのですが，ランダム生成だけでは検査したいテスト・ケースをすべて網羅することが一般的には困難です．

● ダイレクト・テストへの切り替え

ランダム生成でうまくいかない場合には，テストベンチをダイレクト・テストに切り替えます．

ランダム生成を中心にテストベンチを組んだ場合，後からダイレクト・テストに切り替えることは，テストベンチを変更しなければならず，大変な労力が必要になるように感じます．しかし，VMMでは，ランダム生成のテストベンチが出来上がっていると，その枠組みでダイレクト・テストのテストベンチとして使用できます．

図5に示すようにvmm_atomic_genでランダム生成を行う環境(xy_dat_atomic_gen)を構築しておくと，injectというメソッドがあらかじめ組み込まれています．vmm_atomic_genでは，ランダム生成を行い，その結果をチャネルに出力するのですが，injectを使うと横からデータをチャネルに投げることができます．このため，ダイレクト・テストを行いたい場合，データ値を検証者が設定し，injectを呼び出します．こうすることで，検証者が指定した値をチャネルに投げられます．

リスト8にinjectを使った例を示します．vmm_atomic_genによって作成したランダム生成のクラスxy_dat_atomic_genの持つinjectを用いてチャネルに投げている記述です．injectの引き数としては，

図4　外部に記述した制約を別ファイルに

特に，外部に記述した制約を別ファイルにしておけば，ファイルを入れ替えることで制約を変更できる．
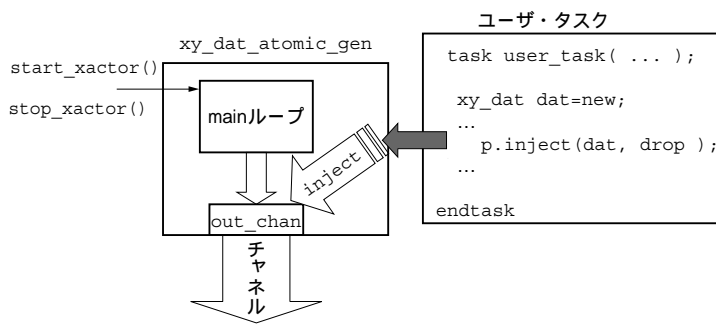
図5　ランダム生成とダイレクト生成

VMMのランダム生成atomic_genは，ランダム生成を行う仕組みに加えて，ダイレクト・テストを行うための環境が組み込まれている．

リスト8　ダイレクト生成

xy_dat_atomic_genのinjectを利用することで，任意の値をチャネルに出力できる．テストベンチの起動時に，ランダム生成を起動せずに，タスクdirectを呼ぶとダイレクト・テストを行うことになる．

```
  task direct(xy_dat_atomic_gen p);
   bit drop;
   int    i;
   xy_dat dat=new;
   for(i=0; i<20 ; i++)  begin
     dat.mX =i;
     dat.mY =i+1;
     p.inject( dat,drop );
   end
 endtask
```

（a）injectの利用

```
  virtual task start();
     super.start();
      xy_drv.start_xactor();
//      xy_gen.start_xactor();
      direct(xy_gen);

  endtask
```

（b）ダイレクト・テスト実行

リスト9　コールバック・クラスの作成

vmm_atomic_genでランダム生成を作成すると，自動的にコールバック・クラスが用意される．そのクラスを継承して，必要な機能を実現できる．

```
class my_xy_dat_atomic_gen_callbacks extends xy_dat_atomic_gen_callbacks;
 virtual task post_inst_gen(xy_dat_atomic_gen gen, xy_dat  data, ref  bit drop);
   string  strX, strY, str;
   strX.itoa(data.mX);
   strY.itoa(data.mY);
   str = {"(", strX, ",", strY ,")"};
   'vmm_note(data.log, str);
 endtask
endclass
```

xy_datのmXとmYの値を指定することで，その値をチャネルに投げることになります．

このinjectを使うときには，ランダム生成xy_dat_atomic_genは停止していることが望ましいので，vmm_envの実行シーケンスのスタート時に，start_xactorを実行せずにコメント・アウトし，その代わりに，ユーザ・タスクであるdirectを呼び出してダイレクト・テストを実行しています．

● ランダム生成後に任意の処理を行う

VMMのランダム生成を使ったときに，いくつか不満があります．例えば，ランダム生成の結果を出力したい場合などに，受信側（チャネルの受け側）でしか確認できません．しかし，ランダム生成したときに値をログに出力したい場合や，データの値を変更したい場合などは，どのようにしたらよいでしょうか．そのために，コールバックという手法が利用できることになっています．

vmm_atomic_genから作成したxy_dat_atomic_genには，xy_dat_atomic_gen_callbacksというコールバック用のクラスが自動的に用意されています．そのクラスでは，post_inst_genというvirtualなタスクが用意されています．このクラスを継承して，新しいpost_inst_genというタスクを定義（関数のオーバライド）し，そのコールバック用のクラスを追加することで，ランダム生成後に任意の処理を行えます．

リスト10　コールバック・クラスの組み込み

コールバック・クラスを組み込むには，ランダム生成をインスタンスした時に，コールバック・クラスもインスタンスし，append_callbackで追加する．

```
 class xy_env extends vmm_env;
  xy_dat_channel     g_d_chan;
  xy_dat_atomic_gen xy_gen;
  xy_drive20         xy_drv;
  my_xy_dat_atomic_gen_callbacks  xy_atm_clb;
   ・・・
  virtual function void build();
     super.build();
     g_d_chan=new("Gen_drv_channel","u0");
     xy_gen  =new("u1",,g_d_chan);
     xy_atm_clb = new;
     xy_gen.append_callback(xy_atm_clb);
     xy_drv = new("u2",,g_d_chan);
  endfunction
```

リスト9に示すのが，コールバック用のクラスを継承して新しいクラスを作成したものです．post_inst_genでは第2引き数でデータが渡されることになっており，この例ではデータの値をvmm_noteを用いてメッセージに出力しています．コールバック用のクラスは定義しただけでは組み込まれませんので，vmm_env（を継承したクラス）で実際に組み込む必要があります．

リスト10で，コールバック用のクラスを実際にxy_dat_atomic_genの変数xy_genに組み込んでいます．組み込むときには，コールバック用のクラスをインスタンスし，そのインスタンスをxy_genにappend_

callbacksするという方法を採ります．また，コールバックは複数追加することができるので，コールバックを利用することで，さまざまな処理を追加できます．

ここでのポイントは，このコールバックはvmm_envのレベルで追加できることです．VMMでは多くの機能追加・変更はvmm_envのレベルで行います．これにより，テストベンチのほとんどの部分を変更しなくてよいという特徴があります．

● ランダム生成を制御する

ランダム生成では，デフォルトでは無限個のデータをランダム生成しようとします．しかし実際には，データをいくつ流すかを指定するケースが多くなります．そのため，vmm_atomic_genでは，stop_after_n_instsという変数を持っています．stop_after_n_instsに生成したいランダム生成の数を指定すると，指定した数だけランダム生成して停止します．

このようにランダム生成ひとつとっても，ランダム生成のON/OFF，制約のON/OFF，制約の書き方，コールバックなどがあります．ぜひ，VMMを読んで，その中に書かれた内容を理解し，日ごろの業務に適用していただければと思います．

参考・引用＊文献

(1) Janic Bergeron, Eduard Cerny, Alan Hunter, Andrew Nightingale著，STARC, ARM, Synopsys監訳；ベリフィケーション・メソドロジ・マニュアル，CQ出版社，2006年4月

あかぼし ひろき
(株)ソリューション・デザイン・ラボラトリ

<筆者プロフィール>
赤星博輝．ハードウェアの検証とソフトウェアのテストの融合が現在のテーマです．ハードウェアではVerification Methodology ManualとSystemVerilogを推進し，ソフトウェアではRTOSを中心に活動中です．